

# Hardware Description Languages: Verilog

- Verilog
  - Structural Models
  - (Combinational) Behavioral Models
  - Syntax
  - Examples

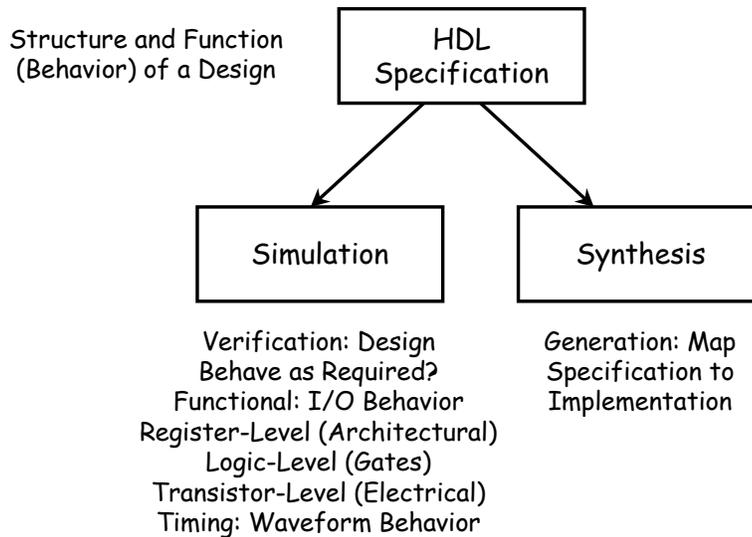
CS 150 - Spring 2007 - Lecture #4: Verilog - 1

## Quick History of HDLs

- ISP (circa 1977) - research project at CMU
  - Simulation, but no synthesis
- Abel (circa 1983) - developed by Data-I/O
  - Targeted to programmable logic devices
  - Not good for much more than state machines
- Verilog (circa 1985) - developed by Gateway (now Cadence)
  - Similar to Pascal and C
  - Delays is only interaction with simulator
  - Fairly efficient and easy to write
  - IEEE standard
- VHDL (circa 1987) - DoD sponsored standard
  - Similar to Ada (emphasis on re-use and maintainability)
  - Simulation semantics visible
  - Very general but verbose
  - IEEE standard

CS 150 - Spring 2007 - Lecture #4: Verilog - 2

# Design Methodology



CS 150 - Spring 2007 - Lecture #4: Verilog - 3

# Verilog/VHDL

- The "standard" languages
- Very similar
  - Many tools provide front-ends to both
  - Verilog is "simpler"
    - Less syntax, fewer constructs
  - VHDL supports large, complex systems
    - Better support for modularization
    - More grungy details
    - "Hello world" is much bigger in VHDL

CS 150 - Spring 2007 - Lecture #4: Verilog - 4

# Verilog

- Supports structural and behavioral descriptions
- Structural
  - Explicit structure of the circuit
  - How a module is composed as an interconnection of more primitive modules/components
  - E.g., each logic gate instantiated and connected to others
- Behavioral
  - Program describes input/output behavior of circuit
  - Many structural implementations could have same behavior
  - E.g., different implementations of one Boolean function

# Verilog Introduction

- the module describes a component in the circuit
- Two ways to describe:
  - Structural Verilog
    - List of components and how they are connected
    - Just like schematics, but using text
    - Hard to write, hard to decode
    - Useful if you don't have integrated design tools
  - Behavioral Verilog
    - Describe *what* a component does, not *how* it does it
    - Synthesized into a circuit that has this behavior

## Structural Model

- Composition of primitive gates to form more complex module
- Note use of `wire` declaration!

```
module xor_gate (out, a, b);
    input      a, b;
    output     out;
    wire       abar, bbar, t1, t2;

    inverter invA (abar, a);
    inverter invB (bbar, b);
    and_gate and1 (t1, a, bbar);
    and_gate and2 (t2, b, abar);
    or_gate  or1 (out, t1, t2);

endmodule
```

By default, identifiers  
are wires

CS 150 - Spring 2007 - Lecture #4: Verilog - 7

## Structural Model

- Example of full-adder

```
module full_addr (A, B, Cin, S, Cout);
    input  A, B, Cin;
    output S, Cout;

    assign {Cout, S} = A + B + Cin;
endmodule

module adder4 (A, B, Cin, S, Cout);
    input  [3:0] A, B;
    input  Cin;
    output [3:0] S;
    output Cout;
    wire   C1, C2, C3;

    full_addr fa0 (A[0], B[0], Cin, S[0], C1);
    full_addr fa1 (A[1], B[1], C1, S[1], C2);
    full_addr fa2 (A[2], B[2], C2, S[2], C3);
    full_addr fa3 (A[3], B[3], C3, S[3], Cout);
endmodule
```

Behavior

Structural

CS 150 - Spring 2007 - Lecture #4: Verilog - 8

## Simple Behavioral Model

### ■ Combinational logic

- Describe output as a function of inputs
- Note use of `assign` keyword: *continuous assignment*

```
module and_gate (out, in1, in2);  
  input      in1, in2;  
  output     out;  
  
  assign out = in1 & in2;  
  
endmodule
```

Output port of a *primitive* must be first in the list of ports

Restriction does not apply to *modules*

## Verilog Data Types and Values

### ■ Bits - value on a wire

- 0, 1
- x - don't care/don't know
- z - undriven, tri-state

### ■ Vectors of bits

- A[3:0] - **vector of 4 bits**: A[3], A[2], A[1], A[0]
- Treated as an *unsigned integer value*
  - e.g., A < 0??
- Concatenating bits/vectors into a vector
  - e.g., sign extend
  - B[7:0] = {A[3], A[3], A[3], A[3], A[3:0]};
  - B[7:0] = {4{A[3]}, A[3:0]};
- Style: Use a[7:0] = b[7:0] + c;  
Not: a = b + c; // need to look at declaration

# Verilog Numbers

- 14 - ordinary decimal number
- -14 - 2's complement representation
- 12'b0000\_0100\_0110 - binary number with 12 bits (\_ is ignored)
- 12'h046 - hexadecimal number with 12 bits
- Verilog values are *unsigned*
  - e.g., C[4:0] = A[3:0] + B[3:0];
  - if A = 0110 (6) and B = 1010(-6)  
C = 10000 not 00000  
i.e., B is zero-padded, not sign-extended

# Verilog Operators

Verilog Operator	Name	Functional Group
()	bit-select or part-select	
()	parenthesis	
! ~ &   ~& ~  ^ ~^ or ^~	logical negation negation reduction AND reduction OR reduction NAND reduction NOR reduction XOR reduction XNOR	Logical Bit-wise Reduction Reduction Reduction Reduction Reduction Reduction
+ -	unary (sign) plus unary (sign) minus	Arithmetic Arithmetic
{}	concatenation	Concatenation
{}}	replication	Replication
* / %	multiply divide modulus	Arithmetic Arithmetic Arithmetic
+ -	binary plus binary minus	Arithmetic Arithmetic
<< >>	shift left shift right	Shift Shift

> >= < <=	greater than greater than or equal to less than less than or equal to	Relational Relational Relational Relational
== !=	logical equality logical inequality	Equality Equality
=== !==	case equality case inequality	Equality Equality
&	bit-wise AND	Bit-wise
^ ^~ or ~^	bit-wise XOR bit-wise XNOR	Bit-wise Bit-wise
	bit-wise OR	Bit-wise
&&	logical AND	Logical
	logical OR	Logical
?:	conditional	Conditional

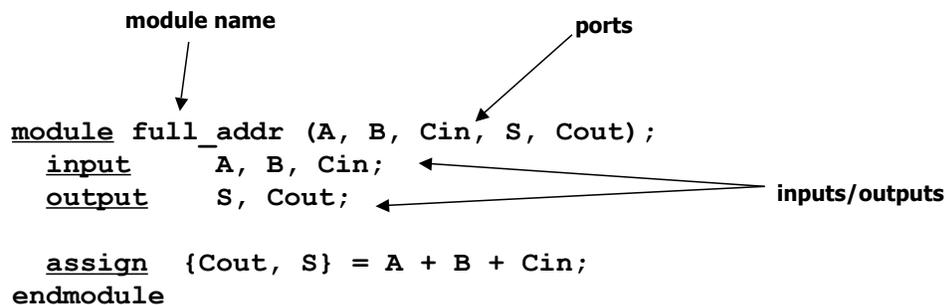
# Verilog "Variables"

- **wire**
  - | Variable used simply to connect components together
- **reg**
  - | Variable that saves a value as part of a behavioral description
  - | Usually corresponds to a wire in the circuit
  - | Is *NOT* necessarily a register in the circuit
- **The Rule:**
  - | Declare a variable as a `reg` if it is the target of a *concurrent (non-blocking)* assignment statement
    - | Don't confuse `reg` assignments with the combinational continuous `assign` statement!
    - | `reg` should only be used with `always` blocks (sequential logic, to be presented ...)
    - | Confusing isn't it?

CS 150 - Spring 2007 - Lecture #4: Verilog - 13

# Verilog Module

- **Corresponds to a circuit component**
  - | "Parameter list" is the list of external connections, aka "ports"
  - | Ports are declared "input", "output" or "inout"
    - | inout ports used on tri-state buses
  - | Port declarations imply that the variables are wires



The diagram shows a Verilog module declaration with annotations. The code is as follows:

```
module full_addr (A, B, Cin, S, Cout);  
  input    A, B, Cin;  
  output  S, Cout;  
  
  assign {Cout, S} = A + B + Cin;  
endmodule
```

Annotations with arrows:

- module name** points to `full_addr`.
- ports** points to the parameter list `(A, B, Cin, S, Cout)`.
- inputs/outputs** points to the `input` and `output` declarations.

CS 150 - Spring 2007 - Lecture #4: Verilog - 14

## Verilog Continuous Assignment

- Assignment is continuously evaluated
- `assign` corresponds to a connection or a simple component with the described function
- Target is *NEVER* a reg variable

```
assign A = X | (Y & ~Z);
```

← use of Boolean operators  
(~ for bit-wise, ! for logical negation)

```
assign B[3:0] = 4'b01XX;
```

← bits can take on four values  
(0, 1, X, Z)

```
assign C[15:0] = 16'h00ff;
```

← variables can be n-bits wide  
(MSB:LSB)

```
assign #3 {Cout, S[3:0]} = A[3:0] + B[3:0] + Cin;
```

← use of arithmetic operator

← multiple assignment (concatenation)

← delay of performing computation, only used by simulator, not synthesis

CS 150 - Spring 2007 - Lecture #4: Verilog - 15

## Comparator Example

```
module Compare1 (A, B, Equal, Alarger, Blarger);
  input    A, B;
  output   Equal, Alarger, Blarger;

  assign Equal = (A & B) | (~A & ~B);
  assign Alarger = (A & ~B);
  assign Blarger = (~A & B);
endmodule
```

## Comparator Example

```
// Make a 4-bit comparator from 4 x 1-bit comparators

module Compare4(A4, B4, Equal, Alarger, Blarger);
  input [3:0] A4, B4;
  output Equal, Alarger, Blarger;
  wire e0, e1, e2, e3, A10, A11, A12, A13, B10, B11, B12, B13;

  Compare1 cp0(A4[0], B4[0], e0, A10, B10);
  Compare1 cp1(A4[1], B4[1], e1, A11, B11);
  Compare1 cp2(A4[2], B4[2], e2, A12, B12);
  Compare1 cp3(A4[3], B4[3], e3, A13, B13);

  assign Equal = (e0 & e1 & e2 & e3);
  assign Alarger = (A13 | (A12 & e3) |
                  (A11 & e3 & e2) |
                  (A10 & e3 & e2 & e1));
  assign Blarger = (~Alarger & ~Equal);
endmodule
```

CS 150 - Spring 2007 - Lecture #4: Verilog - 17

## Announcements

- Lecture room change EFFECTIVE 1 FEB 07:
  - | Beware of what you ask for!
  - | 159 Mulford Hall (near West Gate/Oxford Street)
- Card Key Access to 125 Cory
  - | You can't get it until 7 February
  - | For access:
    - | EECS Keys/Cardkeys, Copy Cards Assistant: Loretta Lutchter
    - | 253 Cory, 642-1527, loret@eecs
    - | Loretta issues keys and electronic cardkeys for Cory Hall.  
Handles cardkey problems. She can add 125 Cory to your Cal Card.

CS 150 - Spring 2007 - Lecture #4: Verilog - 18

## Simple Behavioral Model: the always block

### ■ always block

- Always waiting for a change to a trigger signal
- Then executes the body

```
module and_gate (out, in1, in2);  
  input  in1, in2;  
  output out;  
  reg    out;  
  
  always @(in1 or in2) begin  
    out = in1 & in2;  
  end  
endmodule
```

Not a real register!!  
A Verilog register  
Needed because of  
assignment in always  
block

Specifies when block is executed  
I.e., triggered by which signals

CS 150 - Spring 2007 - Lecture #4: Verilog - 19

## always Block

### ■ Procedure that describes the function of a circuit

- Can contain many statements including *if*, *for*, *while*, *case*
- Statements in the `always` block are executed *sequentially*
  - (Continuous assignments `<=` are executed in *parallel*)
- Entire block is executed at once
- *Final* result describes the function of the circuit for current set of inputs
  - intermediate assignments don't matter, only the final result

### ■ begin/end used to group statements

CS 150 - Spring 2007 - Lecture #4: Verilog - 20

## "Complete" Assignments

- If an `always` block executes, and a variable is *not* assigned
  - Variable keeps its old value (think implicit state!)
  - *NOT* combinational logic  $\Rightarrow$  latch is inserted (implied memory)  
This is usually *not* what you want: dangerous for the novice!
- Any variable assigned in an `always` block should be assigned for any (and every!) execution of the block

## Incomplete Triggers

- Leaving out an input trigger usually results in a sequential circuit
- Example: Output of this "and" gate depends on the input history

```
module and_gate (out, in1, in2);
    input    in1, in2;
    output   out;
    reg     out;

    always @(in1) begin
        out = in1 & in2;
    end

endmodule
```

## Verilog if

### ■ Same as C if statement

```
// Simple 4:1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;    // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;             // target of assignment

    always @(sel or A or B or C or D)
        if (sel == 2'b00) Y = A;
        else if (sel == 2'b01) Y = B;
        else if (sel == 2'b10) Y = C;
        else if (sel == 2'b11) Y = D;

endmodule
```

CS 150 - Spring 2007 - Lecture #4: Verilog - 23

## Verilog if

### ■ Another way

```
// Simple 4:1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;    // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;             // target of assignment

    always @(sel or A or B or C or D)
        if (sel[0] == 0)
            if (sel[1] == 0) Y = A;
            else Y = B;
        else
            if (sel[1] == 0) Y = C;
            else Y = D;

endmodule
```

CS 150 - Spring 2007 - Lecture #4: Verilog - 24

## Verilog case

### ■ Sequential execution of cases

- Only first case that matches is executed (implicit break)
- Default case can be used

```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
  input [1:0] sel;      // 2-bit control signal
  input A, B, C, D;
  output Y;
  reg Y;               // target of assignment

  always @(sel or A or B or C or D)
    case (sel)
      2'b00: Y = A;
      2'b01: Y = B;
      2'b10: Y = C;
      2'b11: Y = D;
    endcase
endmodule
```

↓  
Conditions tested in  
top to bottom order  
↓

CS 150 - Spring 2007 - Lecture #4: Verilog - 25

## Verilog case

### ■ Note:

```
case (case_expression)
  case_item1 : case_item_statement1;
  case_item2 : case_item_statement2;
  case_item3 : case_item_statement3;
  case_item4 : case_item_statement4;
  default   : case_item_statement5;
endcase
```

### ■ ... is the same as ...

```
if(case_expression == case_item1)
  case_item_statement1;
else if (case_expression == case_item2)
  case_item_statement2;
else if (case_expression == case_item3)
  case_item_statement3;
else if (case_expression == case_item4)
  case_item_statement4;
else case_item_statement5;
```

CS 150 - Spring 2007 - Lecture #4: Verilog - 26

## Verilog case

- Without the default case, this example would create a latch for Y
- Assigning X to a variable means synthesis is free to assign any value

```
// Simple binary encoder (input is 1-hot)
module encode (A, Y);
  input  [7:0] A;           // 8-bit input vector
  output [2:0] Y;           // 3-bit encoded output
  reg    [2:0] Y;           // target of assignment

  always @(A)
  case (A)
    8'b00000001: Y = 0;
    8'b00000010: Y = 1;
    8'b00000100: Y = 2;
    8'b00001000: Y = 3;
    8'b00010000: Y = 4;
    8'b00100000: Y = 5;
    8'b01000000: Y = 6;
    8'b10000000: Y = 7;
    default:     Y = 3'bXXX; // Don't care when input is not 1-hot
  endcase
endmodule
```

CS 150 - Spring 2007 - Lecture #4: Verilog - 27

## Verilog case (cont)

- Cases are executed sequentially
  - Following implements a *priority encoder*

```
// Priority encoder
module encode (A, Y);
  input  [7:0] A;           // 8-bit input vector
  output [2:0] Y;           // 3-bit encoded output
  reg    [2:0] Y;           // target of assignment

  always @(A)
  case (1'b1)
    A[0]: Y = 0;
    A[1]: Y = 1;
    A[2]: Y = 2;
    A[3]: Y = 3;
    A[4]: Y = 4;
    A[5]: Y = 5;
    A[6]: Y = 6;
    A[7]: Y = 7;
    default: Y = 3'bXXX; // Don't care when input is all 0's
  endcase
endmodule
```

CS 150 - Spring 2007 - Lecture #4: Verilog - 28

## Parallel Case

- A priority encoder is more expensive than a simple encoder
  - If we know the input is 1-hot, we can tell the synthesis tools
  - "parallel-case" pragma says the order of cases does not matter

```
// simple encoder
module encode (A, Y);
input  [7:0] A;           // 8-bit input vector
output [2:0] Y;           // 3-bit encoded output
reg    [2:0] Y;           // target of assignment

always @(A)
  case (1'b1)             // synthesis parallel-case
    A[0]: Y = 0;
    A[1]: Y = 1;
    A[2]: Y = 2;
    A[3]: Y = 3;
    A[4]: Y = 4;
    A[5]: Y = 5;
    A[6]: Y = 6;
    A[7]: Y = 7;
    default: Y = 3'bX; // Don't care when input is all 0's
  endcase
endmodule
```

CS 150 - Spring 2007 - Lecture #4: Verilog - 29

## Verilog casex

- Like case, but cases can include 'X'
  - X bits not used when evaluating the cases
  - In other words, you don't care about those bits!

## casex Example

```
// Priority encoder
module encode (A, valid, Y);
input  [7:0] A;           // 8-bit input vector
output [2:0] Y;          // 3-bit encoded output
output valid;            // Asserted when an input is not all 0's
reg    [2:0] Y;          // target of assignment
reg    valid;

always @(A) begin
    valid = 1;
    casex (A)
        8'bXXXXXX1: Y = 0;
        8'bXXXXXX10: Y = 1;
        8'bXXXXX100: Y = 2;
        8'bXXXX1000: Y = 3;
        8'bXXX10000: Y = 4;
        8'bXX100000: Y = 5;
        8'bX1000000: Y = 6;
        8'b10000000: Y = 7;
        default: begin
            valid = 0;
            Y = 3'bX; // Don't care when input is all 0's
        end
    endcase
end
endmodule
```

CS 150 - Spring 2007 - Lecture #4: Verilog - 31

## Verilog for

- for is similar to C
- for statement is executed at compile time (like macro expansion)
  - Result is all that matters, not how result is calculated
  - Use in testbenches only!

```
// simple encoder
module encode (A, Y);
input  [7:0] A;           // 8-bit input vector
output [2:0] Y;          // 3-bit encoded output
reg    [2:0] Y;          // target of assignment

integer i;                // Temporary variables for program only
reg [7:0] test;

always @(A) begin
    test = 8b'00000001;
    Y = 3'bX;
    for (i = 0; i < 8; i = i + 1) begin
        if (A == test) Y = i;
        test = test << 1;
    end
end
endmodule
```

CS 150 - Spring 2007 - Lecture #4: Verilog - 32

## Another Behavioral Example

### ■ Computing Conway's Game of Life rule

- Cell with no neighbors or 4 neighbors dies; with 2-3 neighbors lives

```
module life (neighbors, self, out);
  input      self;
  input [7:0] neighbors;
  output     out;
  reg       out;
  integer   count;
  integer   i;

  always @(neighbors or self) begin
    count = 0;
    for (i = 0; i < 8; i = i + 1) count = count + neighbors[i];
    out = 0;
    out = out | (count == 3);
    out = out | ((self == 1) & (count == 2));
  end
endmodule
```

integers are temporary compiler variables

always block is executed instantaneously,  
if there are no delays only the final result is used

CS 150 - Spring 2007 - Lecture #4: Verilog - 33

## Verilog while/repeat/forever

- `while (expression) statement`
  - Execute statement while expression is true
- `repeat (expression) statement`
  - Execute statement a fixed number of times
- `forever statement`
  - Execute statement forever

CS 150 - Spring 2007 - Lecture #4: Verilog - 34

## full-case and parallel-case

### ■ // synthesis parallel\_case

- Tells compiler that ordering of cases is not important
- That is, cases do not overlap
  - e. g., state machine - can't be in multiple states
- Gives cheaper implementation

### ■ // synthesis full\_case

- Tells compiler that cases left out can be treated as don't cares
- Avoids incomplete specification and resulting latches